Homework Set #2
(Data structures, searching and sorting)

Imperative Progamming with Python

January 2015

The homework should be uploaded using the BlackBoard system, it should *not* be printed. Read the `guidelines.pdf` (on the website) for the submission guidelines. **Due date:** 19/01/2015.

Exercise 1 (5 pts.)**.** Write a function `is_sorted(t)` that takes a list and returns `True` if the list is sorted in ascending order and `False` otherwise. You can assume (as a precondition) that the elements of the list can be compared with the relational operators $<$, $>$, etc. For example, `is_sorted([1,2,2])` should return `True` and `is_sorted(['b','a'])` should return `False`.

Exercise 2 (20 pts.)**.** *Searching and sorting.*

a) Write a function `linear_search(e, t)` which looks for element `e` in list `t` by sequentially checking every element, from the first to the last (unless the element is found first). Observe that, if the length of the list is n then this function will take approximately n steps to finish, in the worst-case scenario. This is called the *temporal complexity* of the algorithm.

b) Write a function `sort(t)` which takes a list and sorts it in ascending order ($<$). Observe that the function should modify the parameter and not return a new list. You *cannot* use the sort method of lists nor any built-in sorting function. For this point you should choose one of the following three sorting algorithms and implement it: Bubble sort, Insertion sort, Selection sort. Check (at least) the Wikipedia pages for a description of the algorithms. Observe that these algorithms have a worst-case temporal complexity of approximately n^2.

c) Write a function `binary_search(e, t)` which looks for element `e` in list `t` by using the bisection method described in Exercise 10.11 in the book. The function should return `True` iff the element is found in the list. Observe that, as described in the exercise, the temporal complexity of this algorithm is approximately $\log_2(n)$ where n is the length of the list.

One objective of this exercise, among others, is to show you that sometimes it is useful to keep the data in a special form (in this case, ordered). Suppose you have a database where items are rarely added, but very often asked if an item belongs to the database. In this case, it is worth paying the n^2 price, and sort the database every time an item is added, to be able to work faster –with complexity $\log_2(n)$– in the queries.

Exercise 3 (15 pts.)**.** *Memos.*

a) Read section 11.5 from the book.

b) The *spatial complexity* of an algorithm is the amount of space (memory) it needs to run, with respect to the size of the arguments. In this case, for the fibonacci function, the size of the arguments is n itself. Observe that the implementation given in section 11.5 has a spatial complexity of approximately n (it saves all the values from 0 to n). I know you can do better: Implement a memoizing version of fibonacci with spatial complexity 2. That is, you can only save 2 fibonacci values. Also, you should not use recursion.

That's great! Do you see it? We started with a function which was slow, used a lot of memory and whose call stack was deep and, with some clever tricks, ended up with an iterative, fast version which uses a constant amount of memory no matter which number you want to calculate.

Exercise 4 (30 pts.)**.** *Word prediction and Markov Analysis.*
In this exercise you will analyze text doing (word) frequency analysis and use the data to create a word predictor. That is, you will be able to predict how a partial sentence can be continued (like Google does when you write in the search box). Finally, you will use this prediction schema to generate random text which hopefully has some sense.

a) Read chapter 13 from the book.

b) Do exercises 13.1–13.3, 13.5–13.8 from the book.

Exercise 5 (30 pts.)**.** *Cryptography: Word rotation cipher.*
In this exercise we will develop another cipher. The Caesar cipher of HW1 has many disadvantages, one of them (the least important, probably) is that the output text is *clearly* obfuscated. That is, people would know that you are trying to hide things. A possible way to overcome this, is to rotate *words* instead of characters. This is a good start, but if we want the sentences to make any sense at all we will have to go further and (at least) preserve some grammatical content. In this exercise you are given the following files:

- `nouns.txt`: space-separated list of English nouns.
- `adjectives.txt`: space-separated list of English adjectives.
- `adverbs.txt`: space-separated list of English adverbs.

The encoding procedure is as follows:

1. Choose a random number n, which will be the rotation offset.
2. For each word `w` in the plaintext check if it belongs to nouns, adjectives, or adverbs.
3. If it does not, continue with the next word.
4. If it belongs to the list `t`, let i be the index. Replace the word `w` with `t[(i + n) % len(t)]`.
5. If the word belongs to many lists, just apply the rotation once, with any of the lists.

When applied to plaintext, you should get ciphertext that looks something like this:

> *'What a homologic feeling!' unreportable Alice; 'I must be doberman up like a telescope.'*
> *(Alice in wonderland)*
>
> *"And do you perfunctorily know palaeontological this?" cried Mrs. Gardiner, whose torino rejection to the dramatisation of her casuistry was palaeontologically alive.*
> *(Pride and prejudice)*

a) Write a function `read_file_contents(name)` which, given a filename, returns the contents of the text file, and a function `write_file_contents(name, text)` which, given a filename and some data, writes the (text) data to the file.

b) Write a module `wordrot.py` which includes all the functionality related to the encoding-decoding process. In particular, inside the module, write a function `rotate(s, n)` which takes a string and an integer (that is, positive or negative number) and returns the n-rotation on the string. You will probably also have to write functions which load the nouns, adjectives and adverbs into lists.

c) Write the following two *programs*, which should use (i.e., import, not copy-paste) the functionality from the module `wordrot.py`. Pay attention to preserve the newlines and whitespaces of the original file. That is, if you encode a file and then decode it, you should get exactly the first file.

1. An encoder `encode.py` such that executing '`python encode.py input.txt output.txt`' reads the contents of the file `input.txt`, encode them applying a rotation, print the rotation number, and write the output to the file `output.txt`. Of course these filenames can vary, you should use `sys.argv` to get them.
2. A decoder `decode.py` such that executing '`python decode.py input.txt n output.txt`' reads the contents of the file `input.txt`, decode them applying a n-rotation, and write the output to the file `output.txt`.

Any other combination of arguments in the command line should print 'Arguments invalid.'